04 | WebAssembly 模块的基本组成结构到底有多简单?

2020-09-11 于航

WebAssembly入门课 进入课程 >

讲述：于航
时长 21:00 大小 19.24M

你好，我是于航。今天我来和你聊一聊 Wasm 模块的基本组成结构与字节码分析。

在之前的课程中，我们介绍了 Wasm 其实是一种基于“堆栈机模型” 设计的 V-ISA 指令集。在这节课中，我们将深入 Wasm 模块的字节码结构，探究它在二进制层面的基本布局，以及内部各个结构之间的协作方式。

那为什么要探究 Wasm 在二进制层面的基本布局呢？因为在日常的开发实践中，我们通常只是作为应用者，直接将编译好的 Wasm 二进制模块文件，放到工程中使用就完事了，却很少会去关注 Wasm 在二进制层面的具体组成结构。

但其实只有在真正了解 Wasm 模块的二进制组成结构之后，你才能够知道浏览器引擎在处理和使用一个 Wasm 模块时究竟发生了什么。所以今天我们就将深入到这一部分内容中，

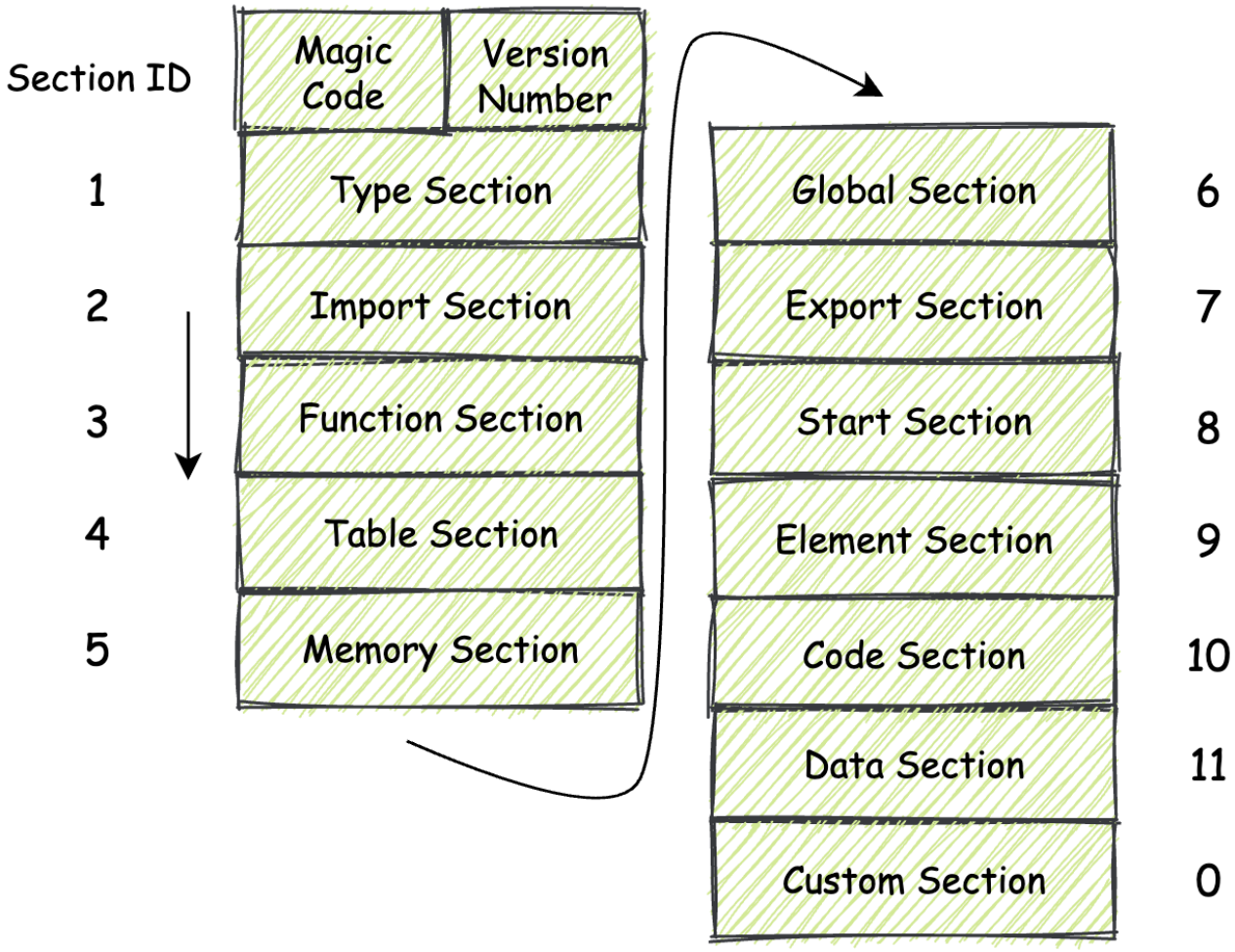
透过现象看本质，为你揭开 Wasm 模块内部组成的真实面目 —— Section。相信通过这一讲，你能够从另一个角度看到 Wasm 的不同面貌。

Section 概览

从整体上来看，同 ELF 二进制文件类似，Wasm 模块的二进制数据也是以 Section 的形式被安排和存放的。Section 翻译成中文是“段”，但为了保证讲解的严谨性，以及你在理解上的准确性，后文我会直接使用它的英文名词 Section。

对于 Section，你可以直接把它想象成，一个个具有特定功能的一簇二进制数据。通常，为了能够更好地组织模块内的二进制数据，我们需要把具有相同功能，或者相关联的那部分二进制数据摆放到一起。而这些被摆放在一起，具有一定相关性的数据，便组成了一个个 Section。

换句话说，每一个不同的 Section 都描述了关于这个 Wasm 模块的一部分信息。而模块内的所有 Section 放在一起，便描述了整个模块在二进制层面的组成结构。在一个标准的 Wasm 模块内，以现阶段的 MVP 标准为参考，可用的 Section 有如下几种。



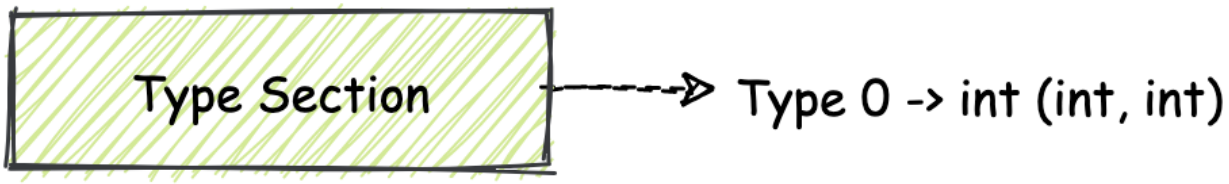
要注意的是，在我们接下来将要讲解的这些 Section 中，除了其中名为 “Custom Secton”，也就是“自定义段”这个 Section 之外，其他的 Section 均需要按照每个 Section 所专有的 Section ID，按照这个 ID 从小到大的顺序，在模块的低地址位到高地址位方向依次进行“摆放”。下面我来分别讲解一下这些基本 Section 的作用和结构。

单体 Section

首先我们来讲解的这部分 Section 被我划分到了“单体 Section”这一类别。也就是说，这一类 Section 一般可以独自描述整个模块的一部分特征（或者说是功能），同时也可以与其他 Section 一起配合起来使用。

当然，这里要强调的是，这样的划分规则只是来源于我自己的设计，希望能够给你在理解 Section 如何相互协作这部分内容时提供一些帮助。这种划分规则并非来源于标准或者官方，你对此有一个概念就好。

Type Section



首先，第一个出现在模块中的 Section 是 “Type Section”。顾名思义，这个 Section 用来存放与“类型”相关的东西。而这里的类型，主要是指“函数类型”。

“函数”作为编程语言的基本代码封装单位，无论是在 C/C++ 这类高级编程语言，还是汇编语言（一般被称为 routine、例程，但也可以理解为函数或者方法）这类低级语言中，都有它的身影，而 Wasm 也不例外。在后面的课程中，我们将会再次详细讲解，如何在浏览器中使用这些被定义在 Wasm 模块内，同时又被标记导出的函数方法，现在你只要先了解这些就可以了。

与大部分编程语言类似，函数类型一般由函数的**参数**和**返回值**两部分组成。而只要知道了这两部分，我们就能够确定在函数调用前后，栈上数据的变化情况。因此，对于“函数类型 “，你也可以将其直接理解为我们更加常见的一个概念 —— “函数签名”。

接下来我们试着更进一步，来看看这个 Section 在二进制层面的具体组成方式。我们可以将 Type Section 的组成内容分为如下两个部分，分别是：所有 Section 都具有的通用“头部”结构，以及各个 Section 所专有的、不同的有效载荷部分。

从整体上来看，每一个 Section 都由有着相同结构的“头部”作为起始，在这部分结构中描述了这个 Section 的一些属性字段，比如不同类型 Section 所专有的 ID、Section 的有效载荷长度。除此之外还有一些可选字段，比如当前 Section 的名称与长度信息等等。关于这部分通用头部结构的具体字段组成，你可以参考下面这张表。

字段	类型	描述
id	varuint7	Section 专有 ID
payload_len	varuint32	该 Section 有效载荷的大小
name_len（可选）	varuint32?	用于自定义段，当 id 为 0 时存在
name（可选）	bytes?	用于自定义段，有效的 UTF8 类型字符串
payload_data	bytes	该 Section 的有效载荷

对于表中第二列给出的一些类型，你目前只需要将它们理解为一种特定的编码方式就可以了，关于这些编码方式和数据类型的具体信息，我会在下一节课中进行讲解。“字段”这一列中的 “name_len” 与 “name” 两个字段主要用于 Custom Section，用来存放这个 Section 名字的长度，以及名字所对应的字符串数据。

对于 Type Section 来说，它的专有 ID 是 1。紧接着排在“头部”后面的便是这个 Section 相关的有效载荷信息（payload_data）。注意，每个不同类型的 Section 其有效载荷的结构都不相同。比如，Type Section 的有效载荷部分组成如下表所示。

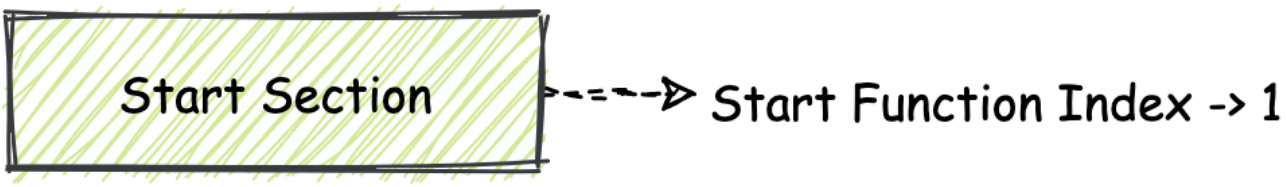
字段	类型	描述
count	varuint32	Type Section 中存放的 Type 个数
entries（多个）	func_type*	连续存放的多个 Type 的实体

可以看到，Type Section 的有效载荷部分是由一个 count 字段和多个 entries 字段数据组合而成的。其中要注意的是 entries 字段对应的 func_type 类型，该类型是一个复合类型，其具体的二进制组成结构又通过另外的一些字段来描述，具体你可以继续参考我下面这张表。

字段	类型	描述
form	varint7	类型构造符 “func” 对应的 OpCode 值
param_count	varuint32	函数的参数个数
param_types（可选）	value_type?	用于有参数的函数，函数的参数类型，依次排列
return_count	varuint1	函数的返回值个数
return_type（可选）	value_type?	用于有返回值的函数，表示返回值类型，依次排列

关于表中各个字段的具体说明，你可以参考表格中最后一列的“描述”信息来进行理解。因为其解读方式与上述的 Section 头部十分类似。更详细的信息，你可以按照需求直接参考官方文档来进行查阅。

Start Section

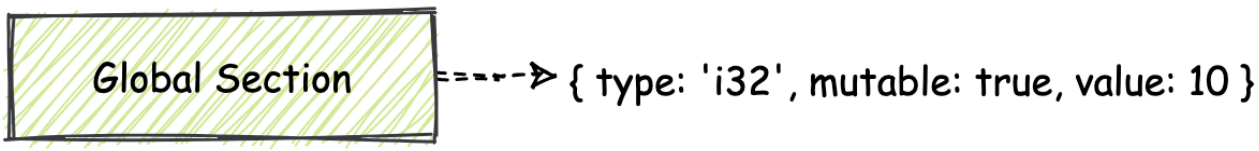


Start Section 的 ID 为 8。通过这个 Section，我们可以为模块指定在其初始化过程完成后，需要首先被宿主环境执行的函数。

所谓的“初始化完成后”是指：模块实例内部的线性内存和 Table，已经通过相应的 Data Section 和 Element Section 填充好相应的数据，但导出函数还无法被宿主环境调用的这个时刻。关于 Data Section 和 Element Section，我们会在下文给你讲解，这里你只需要对它们有一个大致的概念就可以了。

对于 Start Section 来说，有一些限制是需要注意的，比如：一个 Wasm 模块只能拥有一个 Start Section，也就是说只能调用一个函数。并且调用的函数也不能拥有任何参数，同时也不能有任何的返回值。

Global Section



Global Section 的 ID 为 6。同样地，从名字我们也可以猜到，这个 Section 中主要存放了整个模块中使用到的全局数据（变量）信息。这些全局变量信息可以用来控制整个模块的状态，你可以直接把它们类比为我们在 C/C++ 代码中使用的全局变量。

在这个 Section 中，对于每一个全局数据，我们都需要标记出它的值类型、可变性（也就是指这个值是否可以被更改）以及值对应的初始化表达式（指定了该全局变量的初始值）。

Custom Section

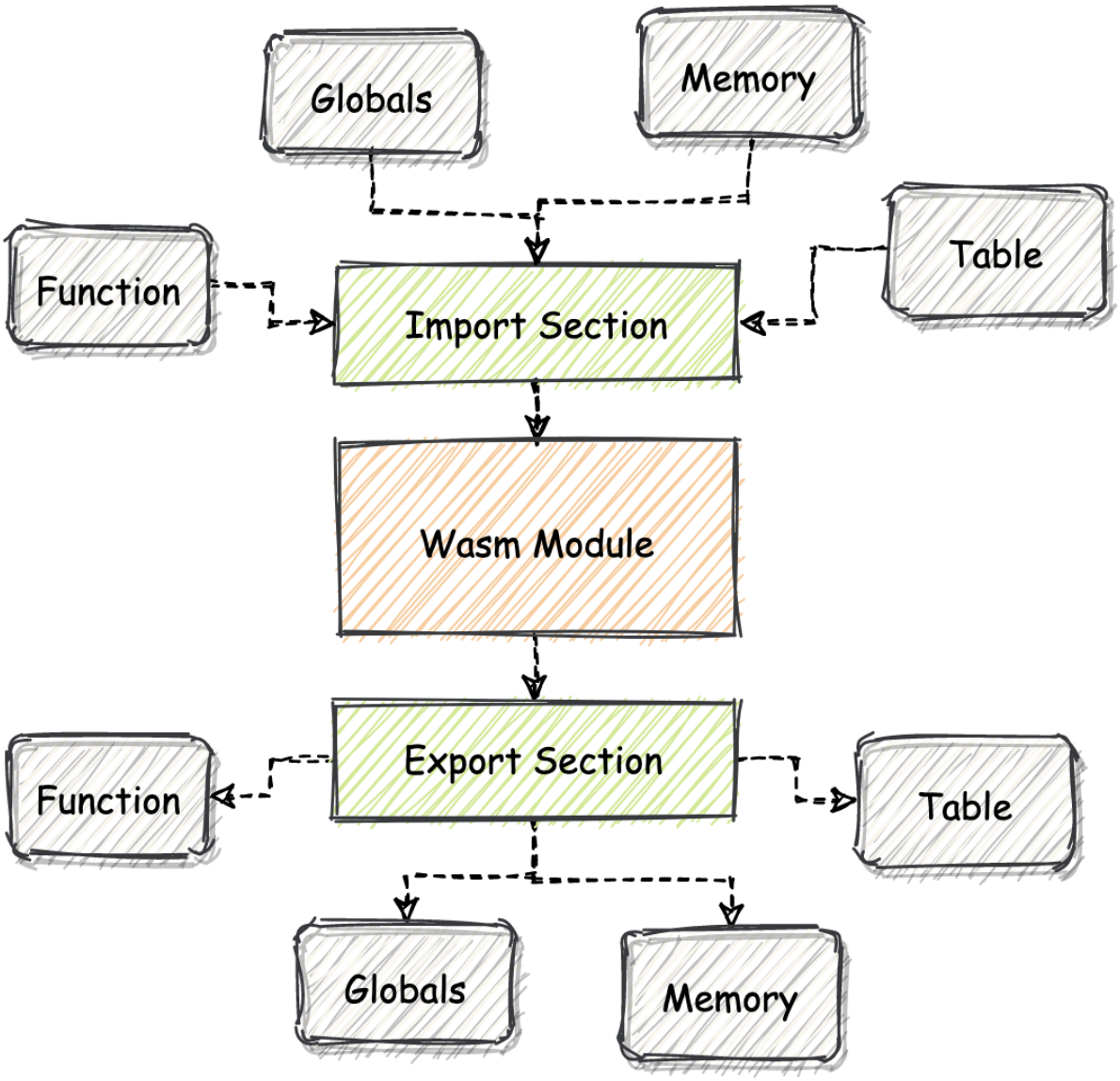
Custom Section 的 ID 为 0。这个 Section 主要用来存放一些与模块本身主体结构无关的数据，比如调试信息、source-map 信息等等。VM（Virtual Machine，虚拟机）在实例化并执行一个 Wasm 二进制模块中的指令时，对于可以识别的 Custom Section，将会以特定的方式为其提供相应的功能。而 VM 对于无法识别的 Custom Section 则会选择直接忽略。

VM 对于 Custom Section 的识别，主要是通过它 “头部”信息中的 “name” 字段来进行。在目前的 MVP 标准中，有且仅有一个标准中明确定义的 Custom Section，也就是 “Name Section”。这个 Section 对应的头部信息中，“name” 字段的值即为字符串 “name”。在这个 Section 中存放了有关模块定义中“可打印名称”的一些信息。

互补 Section

接下来要讲解的这些 Section 被划分到了“互补 Section”这一类别，也就是说，每一组的两个 Section 共同协作，一同描述了整个 Wasm 模块的某方面特征。

Import Section 和 Export Section



为了方便理解，我给你画了张图，你可以通过它来直观地了解这两个 Section 的具体功能。

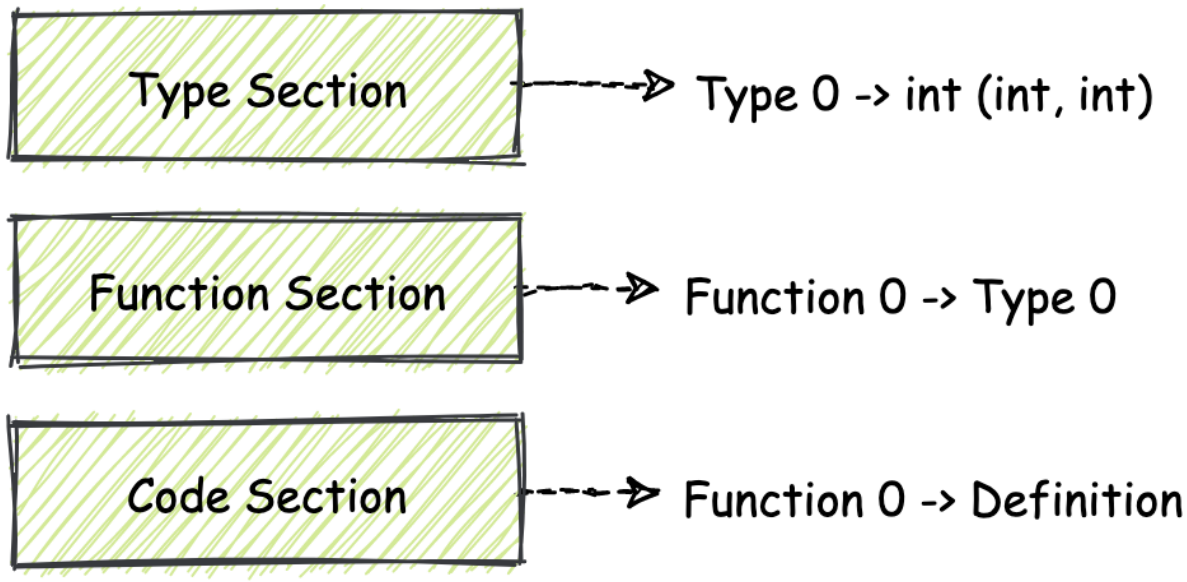
首先是 Import Section，它的 ID 为 2。Import Section 主要用于作为 Wasm 模块的“输入接口”。在这个 Section 中，定义了所有从外界宿主环境导入到模块对象中的资源，这些资源将会在模块的内部被使用。

允许被导入到 Wasm 模块中的资源包括：函数（Function）、全局数据（Global）、线性内存对象（Memory）以及 Table 对象（Table）。那为什么要设计 Import Section 呢？其实就是希望能够在 Wasm 模块之间，以及 Wasm 模块与宿主环境之间共享代码和数据。我将在实战篇中给你详细讲解，如何在浏览器内向一个正在实例化中的 Wasm 模块，导入这些外部数据。

与 Import Section 类似，既然我们可以将资源导入到模块，那么同样地，我们也可以反向地将资源从当前模块导出到外部宿主环境中。

为此，我们便可以利用名为 “Export Section” 的 Section 结构。Export Section 的 ID 为 7，通过它，我们可以将一些资源导出到虚拟机所在的宿主环境中。允许被导出的资源类型同 Import Section 的可导入资源一致。而导出的资源应该如何被表达及处理，则需要由宿主环境运行时的具体实现来决定。

Function Section 和 Code Section



关于 Function Section 与 Code Section 之间的关系，你可以先参考上图，以便有一个直观的印象。Function Section 的 ID 为 3，我想你一定认为，在这个 Section 中存放的是函数体的代码，但事实并非如此。Function Section 中其实存放了这个模块中所有函数对应的函数类型信息。

在 Wasm 标准中，所有模块内使用到的函数都会通过整型的 indicies 来进行索引并调用。你可以想象这样一个数组，在这个数组中的每一个单元格内都存放有一个函数指针，当你需要调用某个函数时，通过 “指定数组下标” 的方式来进行索引就可以了。

而 Function Section 便描述了在这个数组中，从索引 0 开始，一直到数组末尾所有单元格内函数，所分别对应的函数类型信息。这些类型信息是由我们先前介绍的 Type Section 来描述的。

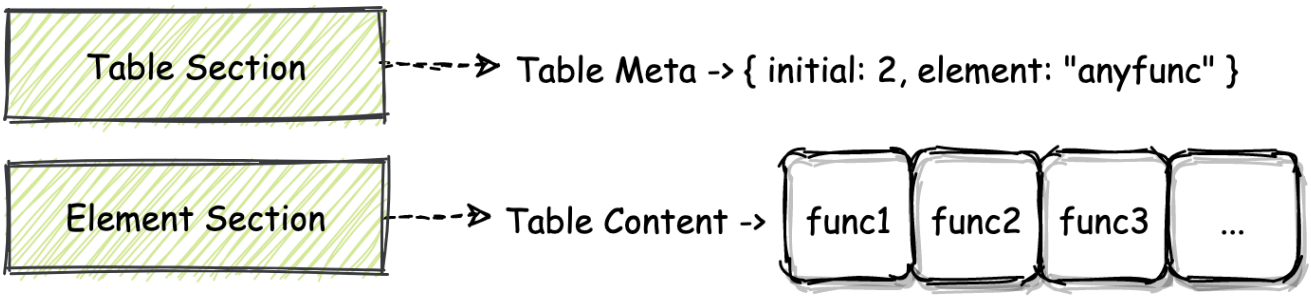
Type Section 存放了 Wasm 模块使用到的所有函数类型（签名）；Function Section 存放了模块内每个函数对应的函数类型，即具体的函数与类型对应关系；而在 Code Section 中存放的则是每个函数的具体定义，也就是实现部分。

Code Section 的 ID 为 10。Code Section 的组织结构从宏观上来看，你同样可以将它理解成一个数组结构，这个数组中的每个单元格都存放着某个函数的具体定义，也就是函数体对应的一簇 Wasm 指令集合。

每个 Code Section 中的单元格都对应着 Function Section 这个“数组”结构在相同索引位置的单元格。也就是说举个例子，Code Section 的 0 号单元格中存放着 Function Section 的 0 号单元格中所描述函数类型对应的具体实现。

当然，上述我们提到的各种“数组”结构，其实并不一定真的是由编程语言中的数组来实现的。只是从各个 Section 概念上的协作和数据引用方式来看，我们可以通过数组来模拟这样的交互流程。具体实现需要依以各个 VM 为准。

Table Section 和 Element Section



同样的，Table Section 与 Element Section 之间的关系，你也可以从上图直观地感受到。Table Section 的 ID 为 4。

在 MVP 标准中，Table Section 的作用并不大，你只需要知道我们可以在其对应的 Table 结构中存放类型为 “anyfunc” 的函数指针，并且还可以通过指令 “call_indirect” 来调用这些函数指针所指向的函数，这就可以了。Table Section 的结构与 Function Section 类似，也都是由“一个个小格子”按顺序排列而成的，你可以用数组的结构来类比着进行理解。

值得说的一点是，在实际的 VM 实现中，虚拟机会将模块的 Table 结构，初始化在独立于模块线性内存的区域中，这个区域无法被模块本身直接访问。因此 Table 中这些“小格

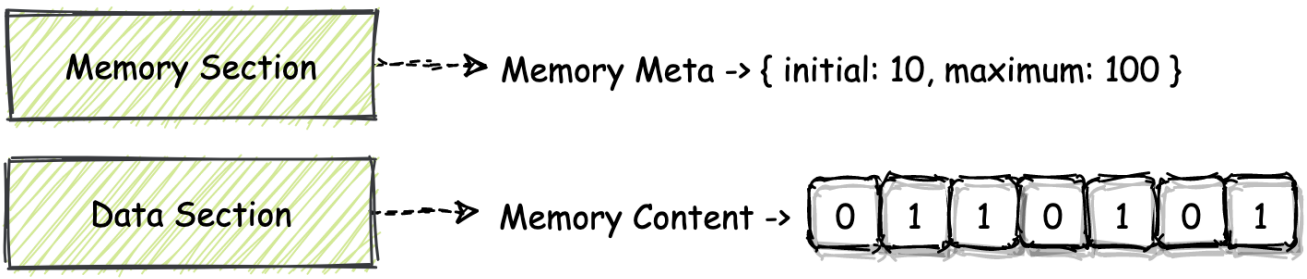
子”内具体存放的值，对于 Wasm 模块本身来说是不可见的。

所以在使用 call_indirect 指令时，我们只能通过 indicies，也就是“索引”的方式，来指定和访问这些“小格子”中的内容。这在某种程度上，保证了 Table 中数据的安全性。

在默认情况下，Table Section 是没有与任何内容相关联的，也就是说从二进制角度来看，在 Table Section 中，只存放了用于描述某个 Table 属性的一些元信息。比如：Table 中可以存放哪种类型的数据？Table 的大小信息？等等。

那为了给 Table Section 所描述的 Table 对象填充实际的数据，我们还需要使用名为 Element Section 的 Section 结构。Element Section 的 ID 为 9，通过这个 Section，我们便可以为 Table 内部填充实际的数据。

Memory Section 和 Data Section



Memory Section 的 ID 为 5。同样，从这个 Section 的名字中我们就基本能够猜到它的用途。同 Table Section 的结构类似，借助 Memory Section，我们可以描述一个 Wasm 模块内所使用的线性内存段的基本情况，比如这段内存的初始大小、以及最大可用大小等等。

Wasm 模块内的线性内存结构，主要用来以二进制字节的形式，存放各类模块可能使用到的数据，比如一段字符串、一些数字值等等。

通过浏览器等宿主环境提供的比如 WebAssembly.Memory 对象，我们可以直接将一个 Wasm 模块内部使用的线性内存结构，以“对象”的形式从模块实例中导出。而被导出的内存对象，可以根据宿主环境的要求，做任何形式的变换和处理，或者也可以直接通过 Import Section ，再次导入给其他的 Wasm 模块来进行使用。

同样地，在 Memory Section 中，也只是存放了描述模块线性内存属性的一些元信息，如果要为线性内存段填充实际的二进制数据，我们还需要使用另外的 Data Section。Data Section 的 ID 为 11。

魔数和版本号

到这里呢，我们就已经大致分析完在 MVP 标准下，Wasm 模块内 Section 的二进制组成结构。但少侠且慢，Section 信息固然十分重要，但另一个更重要的问题是：我们如何识别一个二进制文件是不是一个合法有效的 Wasm 模块文件呢？其实同 ELF 二进制文件一样，Wasm 也同样使用“魔数”来标记其二进制文件类型。所谓魔数，你可以简单地将它理解为具有特定含义 / 功能的一串数字。

一个标准 Wasm 二进制模块文件的头部数据是由具有特殊含义的字节组成的。其中开头的前四个字节分别为 “（高地址）0x6d 0x73 0x61 0x0（低地址）”，这四个字节对应的 ASCII 可见字符为 “asm”（第一个为空字符，不可见）。

接下来的四个字节，用来表示当前 Wasm 二进制文件所使用的 Wasm 标准版本号。就目前来说，所有 Wasm 模块该四个字节的值均为 “（高地址）0x0 0x0 0x0 0x1（低地址）”，即表示版本 1。在实际解析执行 Wasm 模块文件时，VM 也会通过这几个字节来判断，当前正在解析的二进制文件是否是一个合法的 Wasm 二进制模块文件。

在这节课的最后，我们一起来分析一个简单的 Wasm 模块文件的二进制组成结构。这里为了方便你理解，我简化了一下分析流程。我们将使用以下 C/C++ 代码所对应生成的 Wasm 二进制字节码来作为例子进行讲解：

复制代码

```
int add (int a, int b) {
  return a + b;
}
```

在这段代码中，我们定义了一个简单的函数 “add”。这个函数接收两个 int 类型的参数，并返回这两个参数的和。我们使用一个线上的名为 WasmFiddle 的在线 Wasm 编译工具，将上述代码编译成对应的 Wasm 二进制文件，并将它下载到本地。然后，我们可以使用 “hexdump” 命令来查看这个二进制文件的字节码内容。对于这个命令的实际运行结果，你可以参考下面的这张图。

```
➜ Desktop hexdump  -C program.wasm
00000000  00 61 73 6d 01 00 00 00  01 87 80 80 80 00 01 60  |.asm...........`|
00000010  02 7f 7f 01 7f 03 82 80  80 80 00 01 00 04 84 80  |................|
00000020  80 80 00 01 70 00 00 05  83 80 80 80 00 01 00 01  |....p...........|
00000030  06 81 80 80 80 00 00 07  90 80 80 80 00 02 06 6d  |...............m|
00000040  65 6d 6f 72 79 02 00 03  61 64 64 00 00 0a 8d 80  |emory...add.....|
00000050  80 80 00 01 87 80 80 80  00 00 20 01 20 00 6a 0b  |.......... . .j.|
00000060
```

你可以看到，最开始红色方框内的前八个字节 “0x0 0x61 0x73 0x6d 0x1 0x0 0x0 0x0” 便是我们之前介绍的， Wasm 模块文件开头的“魔数”和版本号。这里需要注意地址增长的方向是从左向右。

接下来的 “0x1” 是 Section 头部结构中的 “id” 字段，这里的值为 “0x1” ，表明接下来的数据属于模块的 Type Section。紧接着绿色方框内的五个十六进制数字 “0x87 0x80 0x80 0x80 0x0” 是由 varuint32 编码的 “payload_len” 字段信息，经过解码，它的值为 “0x7” ，表明这个 Section 的有效载荷长度为 7 个字节（关于编解码的具体过程我们会在下一节课中进行讲解）。

根据这节课一开始我们对 Type Section 结构的介绍，你可以知道，Type Section 的有效载荷是由一个 “count” 字段和多个 “entries” 类型数据组成的。因此我们可以进一步推断出，接下来的字节 “0x1” 便代表着，当前 Section 中接下来存在的 “entries” 类型实体的个数为 1 个。

根据同样的分析过程，你可以知道，紧接着紫色方框内的六个十六进制数字序列 “0x60 0x2 0x7f 0x7f 0x1 0x7f” 便代表着 “一个接受两个 i32 类型参数，并返回一个 i32 类型值的函数类型” 。同样的分析过程，也适用于接下来的其他类型 Section，你可以试着结合官方文档给出的各 Section 的详细组成结构，来将剩下的字节分别对应到模块的不同 Section 结构中。

总结

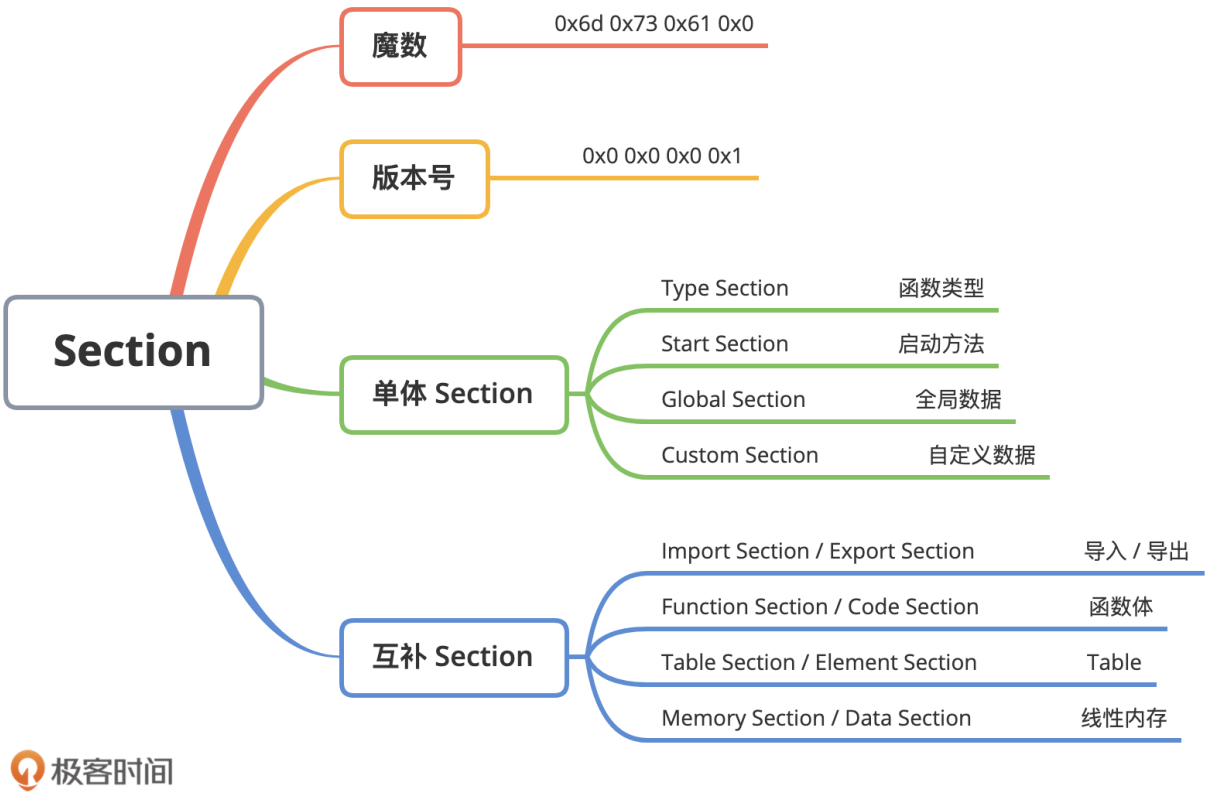
好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我们主要介绍了一个 Wasm 模块在二进制层面的具体组成结构。每一个 Wasm 模块都是由多个不同种类的 Section 组成的，这些 Section 按照其专有 ID 从小到大的顺序被依次摆放着。

其中的一些 Section 可以独自描述 Wasm 模块某个方面的特性，而另外的 Section 则需要与其他类型的 Section 一同协作，来完成对模块其他特性的完整定义。

除了这些专有 Section，模块还可以通过 Custom Section 来支持一些自定义功能。这个 Section 一般可以用于提供一些 VM 专有的、而可能又没有被定义在 Wasm 标准中的功能，比如一些与调试相关的特性等等。

最后，我们还介绍了整个 Wasm 模块中最为重要的，位于模块二进制代码最开始位置的“魔数”以及“版本号”。这两个字段主要会被 VM 用于对 Wasm 模块的类型进行识别，当 VM 检测到二进制文件中的某一个字段不符合规范时，则会立即终止对该模块的初始化和后续处理。这里我放了一张脑图，你可以通过这张图，对 Wasm 模块的整体结构有个更直观的认识。



课后思考

本节课最后，我来给你留一个思考题：

尝试去了解一下 ELF 格式的 Section 结构，并谈谈它与 Wasm Section 在设计上的异同之处？

好，今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我参与讨论，同时欢迎你把这节课分享给你的朋友或者同事，一起交流一下。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | WebAssembly 是一门新的编程语言吗?

下一篇 05 | 二进制编码：WebAssembly 微观世界的基本数据规则是什么?

精选留言 (3)

写留言

大土豆

2020-09-11

ELF是和特定平台相关的二进制格式。WASM是一个通用的字节码格式，和平台无关，由浏览器的解释器执行

5

一步
2020-09-11

能不能结合 wasm 的可识别文本 来讲解各个 section ？ 这个干讲不是特别形象。

比如 使用 assemblyscript 编译后生成的 wat 文件

作者回复: 嗯嗯，收到你的反馈了。主要是考虑到内容的先后顺序，所以这里还是选择从字节码的角度来介绍 Wasm Section 的组成结构。可以先有个大致印象，关于 WAT 的角度，在 06 中会提到哈。

3

xiaobang
2020-09-13

table section 中如果存放的不是anyfunc，调用call indirect 会失败的吧

作者回复: 是的，会抛出 Runtime Exception。anyfunc 作为一个类型，其具体形式是由宿主环境定义的，也就意味着在调用 call_indirect 的时候宿主环境会去检测 Table 中对应位置存放的值是不是可以被作为 anyfunc，如果不是则抛出运行时错误。当然，宿主环境也同样会阻止非 anyfunc 类型被存放在 Table 中。